Practice Midterm 6 Solutions

This page contains solutions to Practice Midterm 6.

Question 1: C++ and ADTs (Tea Descriptions)

This task was mostly about implementing a given specification and demonstrating that you know how to correctly use ADTs, including a nested collection type.

`containsPhrase()`

There are many ways to implement this function, some of which are just subtle variations on the approach below.

If using nested loops, one of the things to be careful of is that we never access vector indices that are out of bounds – especially important when we get to an index in the **`sentence`** vector where there are fewer strings remaining than there are in the **`searchPhrase`** vector.

Some people solved this one by concatenating all the strings in each respective vector into a single corresponding string: one for **`sentence`** and one for **`searchPhrase`**. They then checked whether the phrase string was a substring of the sentence string. The thing to be careful of there is that there must be a reasonable delimiter between each word in the resulting strings (space-separated probably make the most sense) so as not to produce false matches.

```
bool containsPhrase(Vector<string>& sentence, Vector<string>& searchPhrase)
{
  for (int i = 0; i < sentence.size(); i++)
  {
    // We've gotten to a point where the search phrase's length exceeds
    // the number of words remaining in the sentence, so give up.
    if (searchPhrase.size() > sentence.size() - i)
    {
      return false;
    }

    bool match = true;

    for (int j = 0; j < searchPhrase.size(); j++)
    {
      if (sentence[i + j] != searchPhrase[j])
      {
        // Note: We cannot simply return false here. We need to move
        // forward to the next iteration of the outer loop and start
        // out search again.
        match = false;

        // Breaking potentially improves runtime but isn't strictly
        // necessary.
        break;
      }
    }

    if (match)
    {
      return true;
    }
  }

  return false;
}
```

getName()

The goal with this function is to work with a map that is effectively an inversion of the map we would prefer to have at our disposal: what we want to be the keys are actually the values, and the values we need are actually the keys.

```
string getName(Map<string, int>& teaNames, int teaID)
{
    for (string teaName : teaNames)
    {
        if (teaNames[teaName] == teaID)
        {
            return teaName;
        }
    }

    error("Not found.");
}
```

matches()

This part of the problem effectively requires us to loop over the **teaDescriptions** keys, retrieve the vector associated with each of those keys, and loop over the nested vectors within them, relying on our predefined helper functions to solve the problem.

```
Set<string> matches(Map<int, Vector<Vector<string>>>& teaDescriptions,
                    Map<string, int>& teaNames, Vector<string>&
searchPhrase)
{
  Set<string> results;

  for (int id : teaDescriptions)
  {
    Vector<Vector<string>> description = teaDescriptions[id];

    for (Vector<string> sentence : description)
    {
      if (containsPhrase(sentence, searchPhrase))
      {
        results.add(getName(teaNames, id));
        break;  // again, not strictly required
      }
    }
  }

  return results;
}
```

Question 2: Runtime Analysis (Big-O)

This question asked you to give best- and worst-case Big-O runtimes for various functions, as well as to analyze a key shortcoming of one function and to identify best- and worst-case inputs to a couple others.

a) O(n log n) or O(log (n!))

- The O(log (n!)) response is based on the observation that the runtime is effectively log(1) + log(2) + … + log(n) and then applying the following property of logs: log(a) + log(b) = log(ab). Both responses are fine; Stirling's approximation gives us that $\log_2(n!)$ is O(n log n).

b) O(n)

- The set does not allow insertion of duplicates, so inserting the same element repeatedly would result in *n* O(1) operations – for a total runtime of O(n). This is perhaps a bit non-obvious, and so we awarded high partial credit for O(n log n) on this one.

c) O(n log n)

d) O(n log n)

- This corresponds to the case where the vector gets sorted (taking O(n log n) time) but we then return false immediately because `v[0] + 1 != v[1]`.

e) The pass-by-reference vector is ruined (re-ordered) as a side effect of solving the problem.

f) $O(2^n)$

g) O(q)

h) O(q log k)

- Side note: O(log k * q) is a somewhat ambiguous way of writing this solution, because we can't tell whether that's O((log k) * q)) or O(log (k * q)). Parentheses add clarity, but simply reordering the terms as O(q log k) eliminates the ambiguity.

i) A correct answer must meet the following conditions:

- The haystack is sorted.

- The haystack contains 5 elements.
- The needles vector contains 5 copies of the same element.
- The element in the needles vector is in the middle of the haystack vector.
- Example: **`haystack = {1, 2, 3, 4, 5}, needles = {3, 3, 3, 3, 3}`**

j) A correct answer must meet the following conditions:

- The haystack is sorted.
- The haystack contains 5 elements.
- The needles vector contains 5 elements.
- None of the elements in the needles vector are in the haystack.
- Example: **`haystack = {1, 2, 3, 4, 5}, needles = {6, 6, 6, 6, 6}`**
- Side note: Some people populated the needles vector with elements that were in the haystack, but in the positions they thought would take the longest for binary search to reach. Note that searching for elements that are not in the haystack would yield slightly worse runtimes than that because there would be one additional comparison of **`hi`** and **`lo`**.

Question 3: Recursion (isReverse)

This recursive problem required some careful bookkeeping because of the fact that the vectors were passed by reference. There are many possible approaches to the base cases.

```
bool isReverse(Vector<int>& v1, Vector<int>& v2)
{
  if (v1.size() != v2.size())
  {
    return false;
  }

  if (v1.size() == 0)
  {
    return true;
  }

  // Note: You could also hold off on removing these integers from the
  // vectors until after the return false case below.
  int v1removed = v1.remove(0);
  int v2removed = v2.remove(v2.size() - 1);

  bool result;

  // We don't return within the following blocks because we still need
  // to add the removed elements back to the vectors.
  if (v1removed != v2removed)
  {
    result = false;
  }
  else
  {
    result = isReverse(v1, v2);
  }

  // Add elements back to the specific positions we removed them from.
  v1.insert(0, v1removed);
  v2.add(v2removed);

  return result;
}
```

Note that there are many possible ways to approach the base cases. Do you see why each of the following works?

```
// Base Case Solution 1
// ====================
// Note: We can't swap the order of these base cases unless we modify
// the size == 0 check to also look at v2.size(). See note below.
if (v1.size() != v2.size())
{
  return false;
}
if (v1.size() == 0)
{
  return true;
}

// Base Case Solution 2
// ====================
// Note: If checking for size == 0 before checking for a size mismatch,
// we have to check that BOTH v1.size() and v2.size() are 0 before
// returning true, like so:
if (v1.size() == 0 && v2.size() == 0)
{
  return true;
}
if (v1.size() != v2.size())
{
  return false;
}

// Base Case Solution 3
// ====================
// Note: An alternative to the base cases above would be to check if
// BOTH vectors are empty (return true) and otherwise recurse until
// ONE is empty but not the other (return false). This is potentially
// slower than the approaches above in cases where we have long vectors
// whose lengths are mismatched from the start (because we have to
// whittle away at them, one by one, until one of the vectors ends up
// empty), but this is still a valid approach.
//
// More explicitly, the alternative base cases are as given below.
//
// The use of the || operator below might seem wonky at first. If we get to
// that second condition, we know the sizes aren't BOTH zero, which means
// that if one or the other is zero, it must actually be the case that
// EXACTLY one or the other is zero. (At least one is NOT zero if we reach
// that second if statement.)
if (v1.size() == 0 && v2.size() == 0)
{
  return true;
}
if (v1.size() == 0 || v2.size() == 0)
{
  return false;
}
```

Question 4: Recursive Fractal Tracing

This problem involved tracing the order in which calls were made (and the order in which they

Getting to implement a function for drawing a fractal. This closely mirrored one of the lecture quiz questions.

- Order of calls being made (out of B, C, D, E, F, G, and H):
  - First call: **B**
  - Second call: **E**
  - Third call: **F**
  - Fourth call: **C**
  - Last call: **H**
- Order of calls returning (out of B, C, D, E, F, G, and H):
  - First return: **E**
  - Second return: **F**
  - Third return: **B**
  - Fourth return: **G**
  - Last return: **D**

Note that **A** was excluded from the menu of options because the problem already stated it was the first call to be made.

Question 5: Problem Solving with ADTs

This problem involved coming up with an iterative solution to a problem we had solved recursively in class. The main idea behind this algorithm (as mentioned in the problem write-up) is similar to the one described in A2 to perform BFS: we take each proto-solution from a queue, add to it all possible "moves" that could come next (in this case, that's always "H" and "T", because those are the possible outcomes for our next coin flip), and put the newly extended proto-solutions back into the queue.

There are a few ways to approach this problem, some more exotic than others.

Solution 1

This solution uses a single queue. For each string we dequeue, we enqueue two new ones (adding "H" and "T" to the dequeued string).

```
void printCoinFlips(int n)
{
  Queue<string> q;

  q.enqueue("");

  for (int i = 1; i <= n; i++)
  {
    int ogSize = q.size();

    for (int j = 0; j < ogSize; j++)
    {
      string s = q.dequeue();

      q.enqueue(s + "H");
      q.enqueue(s + "T");
    }
  }

  cout << q << endl;
}
```

Solution 2

This solution is similar to the one above but uses two queues to slightly simplify the looping condition as we empty the first queue.

```
void printCoinFlips(int n)
{
  Queue<string> q;
  Queue<string> tempQ;
  q.enqueue("");

  for (int i = 1; i <= n; i++)
  {
    // We can use this looping condition because we're not
    // adding elements to this queue within the loop.
    while (!q.isEmpty())
    {
      string s = q.dequeue();

      tempQ.enqueue(s + "H");
      tempQ.enqueue(s + "T");
    }

    // Now dump all the results from the tempQ back into q.
    while (!tempQ.isEmpty())
    {
      q.enqueue(tempQ.dequeue());
    }
  }

  cout << q << endl;
}
```

Solution 3

This solution is based on the observation that our final output needs to have 2^n strings.

```
void printCoinFlips(int n)
{
  Queue<string> q = {""};
  int numResults = pow(2, n);

  for (int i = 1; i < numResults; i++)
  {
    string current = q.dequeue();
    q.enqueue(current + "H");
    q.enqueue(current + "T");
  }

  cout << q << endl;
}
```

Solution 4

This approach keeps extending strings until they reach the desired length, at which time it prints them out and stops adding new, longer strings to the queue.

There is no output for this function for n = 0, but that's fine because the problem description said we weren't super picky about output formatting. One could consider the lack of output to be the equivalent of printing an empty string to the string, and for n = 0, that's the correct result.

```
void printCoinFlips(int n)
{
  if (n == 0)
  {
    return;
  }
  Queue<string> q = {"H", "T"};
  while (!q.isEmpty())
  {
    string latest = q.dequeue();
    if (latest.length() == n)
    {
      cout << latest << endl;
    }
    else
    {
      q.enqueue(latest + "H");
      q.enqueue(latest + "T");
    }
  }
}
```


Website programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-Apr-21